Appendix A 8088 INSTRUCTION SET

INTRODUCTION

The 8086 and 8088 execute exactly the same instructions. This instruction set includes equivalents to the instruction typically found in previous microprocessors, such as the 8080/8085. Significant new operations include:

- Multiplication and division of signed and unsigned binary numbers as well as unpacked decimal numbers
- Move, scan, and compare operations for strings up to 64K bytes in length
- Nondestructive bit testing
- Byte translation from one code to another
- Software-generated interrupts
- A group of instructions that can help coordinate the activities of multiprocessor systems

These instructions treat different types of operands uniformly. Nearly every instruction can operate on either byte or word data. Register, memory, and immediate operands may be specified interchangeably in most instructions (except, of course, that immediate values may only serve as source and not destination operands). In particular, memory variables can be added to, subtracted from, shifted, compared, and so on, in place, without moving them in and out of registers. This saves instructions, registers, and execution time in assembly language programs. In high-level languages, where most variables are memory based, compilers, such as PL/M-86, can produce faster and shorter object programs.

The 8086/8088 instruction set can be viewed as existing at two levels: the assembly level and the machine level. To the assembly language programmer, the 8086 and 8088 appear to have a repertoire of about 100 instructions. One MOV (move) instruction, for example, transfers a byte or a word from a register or a memory location or an immediate value to either a register or a memory location. The 8086 and 8088 CPUs, however, recognize 28 different MOV machine instructions ("move byte register to memory," "move word immediate to register," etc.). The ASM-86 assembler translates the assembly-level instructions written by a programmer into the machine-level instructions that are actually executed by the 8086 or 8088. Compilers such as PL/M-86 translate high-level language statements directly into machine-level instructions.

The two levels of the instruction set address two different requirements: efficiency and simplicity. The numerous—there are about 300 in all—forms of machine-level instructions allow these instructions to make very efficient use of storage. For example, the

machine instruction that increments a memory operand is three or four bytes long because the address of the operand must be encoded in the instruction. To increment a register, however, does not require as much information, so the instruction can be shorter. In fact, the 8086 and 8088 have eight different machine-level instructions that increment a different 16-bit register; these instructions are only one byte long.

If a programmer had to write one instruction to increment a register, another to increment a memory variable, etc., the benefit of compact instructions would be offset by the difficulty of programming. The assembly-level instructions simplify the programmer's view of the instruction set. The programmer writes one form of the INC (increment) instruction and the ASM-86 assembler examines the operand to determine which machine-level instruction to generate.

This section presents the 8086/8088 instruction set from two perspectives. First, the assembly-level instructions are described in functional terms. The assembly-level instructions are then presented in a reference table that breaks out all permissible operand combinations with execution times and machine instruction length, plus the effect that the instruction has on the CPU flags.

DATA TRANSFER INSTRUCTIONS

The 14 data transfer instructions (Table A-1) move single bytes and words between memory and register as well as between register AL or AX and I/O ports. The stack manipulation instructions are included in this group as are instructions for transferring flag contents and for loading segment registers.

Table A-1: Data Transfer Instructions

	GENERAL PURPOSE
MOV	Move byte or word
PUSH	Push word onto stack
POP	Pop word off stack
XCHG	Exchange byte or word
XLAT	Translate byte
	INPUT/OUTPUT
IN	Input byte or word
OUT	Output byte or word
	ADDRESS OBJECT
LEA	Load effective adress
LDS	Load pointer using DS
LES	Load pointer using ES
	FLAG TRANSFER
LAHF	Load AH register from flags
SAHF	Store AH register in flags
PUSHF	Push flags onto stack
POPF	Pop flags off stack

GENERAL PURPOSE DATA TRANSFERS

MOV *destination, source*

MOV transfers a byte or a word from the source operand to the destination operand.

PUSH *source*

PUSH decrements SP (the stack pointer) by two and then transfers a word from the source operand to the top of stack now pointed by SP. PUSH often is used to place parameters on the stack before calling a procedure; more generally, it is the basic means of storing temporary data on the stack.

POP *destination*

POP transfers the word at the current top of stack (pointed to by SP) to the destination operand, and then increments SP by two to point to the new top of stack. POP can be used to move temporary variables from the stack to registers or memory.

XCHG *destination, source*

XCHG (exchange) switches the contents of the source and destination (byte or word) operands. When used in conjunction with the LOCK prefix, XCHG can test and set a semaphore that controls access to a resource shared by multiple processors.

XLAT *translate-table*

XLAT (translate) replaces a byte in the AL register with a byte from a 256-byte, user-coded translation table. Register BX is assumed to point to the beginning of the table. The byte in AL is used as an index into the table and is replaced by the byte at the offset in the table corresponding to AL's binary value. The first byte in the table has an offset of 0. For example, if AL contains 5H, and the sixth element of the translation table contains 33H, then AL will contain 33H following the instruction. XLAT is useful for translating characters from one code to another, the classic example being ASCII to EBCDIC or the reverse.

IN *accumulator, port*

IN transfers a byte or a word, respectively, to the AL register or AX register, from an input port. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through 255, or with a number previously placed in the DX register, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535.

OUT *port, accumulator*

OUT transfers a byte or a word from the AL register or the AX register, respectively, to an output port. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through 255, or with a number previously placed in register DDX, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535).

ADDRESS OBJECT TRANSFERS

These instructions manipulate the addresses of variables rather than the contents or values of variables. They are most useful for list processing, based variables, and string operations.

LEA *destination, source*

LEA (Load Effective Address) transfers the offset of the source operand (rather than its value) to the destination operand. The source operand must be a memory operand, and the destination operand must be a 16-bit general register. LEA does not affect any flags. The XLAT and string instructions assume that certain registers point to operands; LEA can be used to lead these registers (e.g., loading BX with the address of the translate table used by the XLAT instruction).

LDS *destination, source*

LDS (Load pointer using DS) transfers a 32-bit pointer variable from source operand, which must be a memory operand, to the destination operand and register DS. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the pointer is transferred to register DS. Specifying SI as the destination operand is a convenient way to prepare to process a source string that is not in the current data segment (string instructions assume that the source string is located in the current data segment and that SI contains the offset of the string).

LES *destination, source*

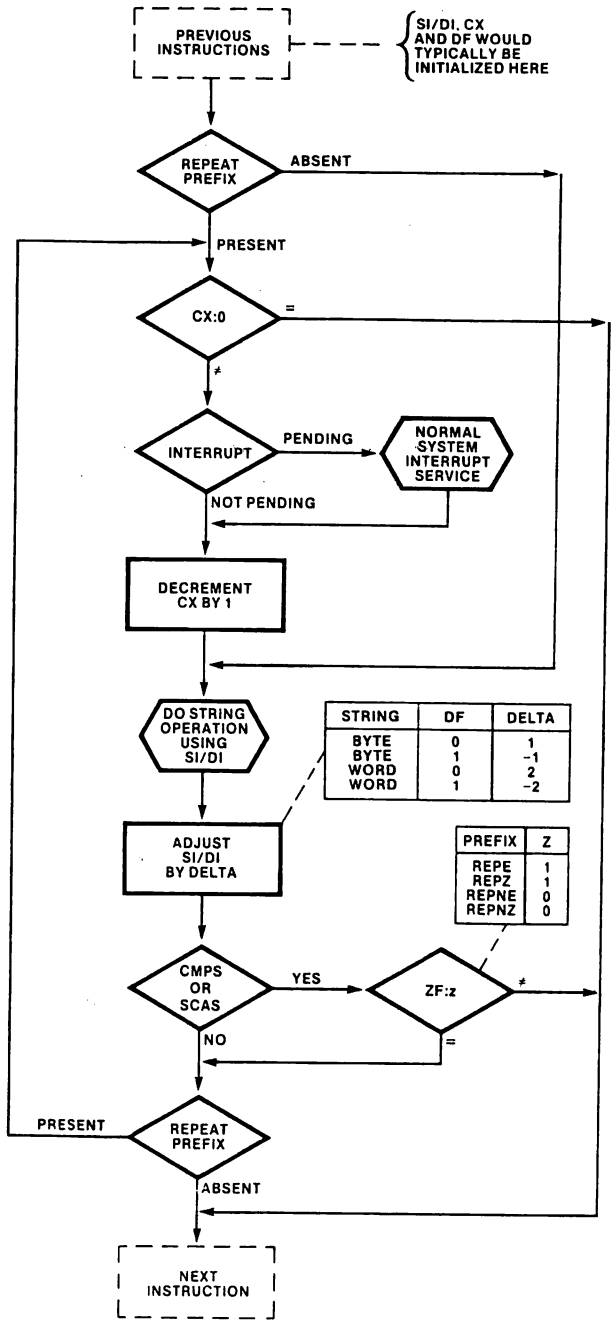
LES (Load pointer using ES) transfers a 32-bit pointer variable from the source operand, which must be a memory operand, to the destination operand and register ES. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the pointer is transferred to register ES. Specifying DI as the destination operand is a convenient way to prepare to process a destination string that is not in the current extra segment. (The destination string must be located in the extra segment, and DI must contain the offset of the string.)

FLAG TRANSFERS

LAHF

LAHF (Load register AH from Flags) copies SF, ZF, AF, PF and CF (the 8080/8085 flags) into bits 7, 6, 4, 2 and 0, respectively, of register AH (see Figure A-1). The content of bits 5, 3 and 1 is undefined; the flags themselves are not affected. LAHF is provided primarily for converting 8080/8085 assembly language programs to run on an 8086 or 8088.

Figure A-1: String Operation Flow



SAHF

SAHF (Store register AH into Flags) transfers bits 7, 6, 4, 2, and 0 from register AH into SF, ZF, AF, PF, and CF, respectively, replacing whatever values these flags previously had. OF, DF, IF and TF are not affected. This instruction is provided for 8080/8085 compatibility.

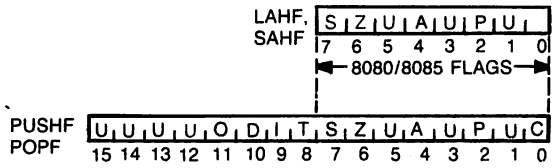
PUSHF

PUSHF decrements SP (the stack pointer) by two and then transfers all flags to the word at the top of stack pointed to be SP (see Figure A-1). The flags themselves are not affected.

POPF

POPF transfers specific bits from the word at the current top of stack (pointed to by register SP) into the 8086/8088 flags, replacing whatever values the flags previously contained (Figure A-2). SP is then incremented by two to point to the new top of stack. PUSHF and POPF allow a procedure to save and restore a calling program's flags. They also allow a program to change the setting of TF (there is no instruction for updating this flag directly). The change is accomplished by pushing the flags, altering bit 8 of the memory image, and then popping the flags.

Figure A-2: Flag Storage Formats



U = UNDEFINED; VALUE IS INDETERMINATE
O = OVERFLOW FLAG
D = DIRECTION FLAG
I = INTERRUPT ENABLE FLAG
T = TRAP FLAG
S = SIGN FLAG
Z = ZERO FLAG
A = AUXILIARY CARRY FLAG
P = PARITY FLAG
C = CARRY FLAG

ARITHMETIC INSTRUCTIONS

ARITHMETIC DATA FORMATS

8086 and 8088 arithmetic operations (Table A-2) may be performed on four types of numbers: unsigned binary, signed binary (integers), unsigned packed decimal and unsigned unpacked decimal (see Table A-3). Binary numbers may be 8 or 16 bits long. Decimal numbers are stored in bytes, two digits per byte for packed decimal and one digit per byte for unpacked decimal. The processor always assumes that the operands specified in arithmetic instructions contain data that represent valid numbers for the type of instruction being performed. Invalid data may produce unpredictable results.

Table A-2: Arithmetic Instructions

	ADDITION
ADD	Add byte or word
ADC	Add byte or word with carry
INC	Increment byte or word by 1
AAA	ASCII adjust for addition
DAA	Decimal adjust for addition
	SUBTRACTION
SUB	Subtract byte or word
SBB	Subtract byte or word with borrow.
DEC	Decrement byte or word by 1
NEG	Negate byte or word
CMP	Compare byte or word
AAS	ASCII adjust for subtraction
DAS	Decimal adjust for subtraction
	MULTIPLICATION
MUL	Multiply byte or word unsigned
IMUL	Integer multiply byte or word
AAM	ASCII adjust for multiply
	DIVISION
DIV	Divide byte or word unsigned
IDIV	Integer divide byte or word
AAD	ASCII adjust for division
CBW	Convert byte to word
CWD	Convert word to doubleword

Table A-3: Arithmetic Interpretation of 8-Bit Numbers

HEX	BIT PATTERN	UNSIGNED BINARY	SIGNED BINARY	UNPACKED DECIMAL	PACKED DECIMAL
07	0 0 0 0 0 1 1 1	7	+7	7	7
89	1 0 0 0 1 0 0 1	137	-119	Invalid	89
C5	1 1 0 0 0 1 0 1	197	-59	Invalid	Invalid

Unsigned binary numbers may be either 8 or 16 bits long; all are considered in determining a number's magnitude. The value range of an 8-bit unsigned binary number is 0-255; 16 bits can represent values from 0 through 65,535. Addition, subtraction, multiplication, and division operations are available for unsigned binary numbers.

Signed binary numbers (integers) may be either 8 or 16 bits long. The high-order (leftmost) bit is interpreted as the number's sign: 0 = positive, and 1 = negative. Negative numbers are represented in standard two's complement notation. Since the high-order bit is used for a sign, the range of an 8-bit integer is -128 through +127; 16-bit integers may range from -32,768 through +32,767. The value zero has a positive sign. Multiplication and division operations are provided for signed binary numbers. Addition and subtraction are performed with the unsigned binary instructions. Conditional jump instructions, as well as an "interrupt on overflow" instruction, can be used following an unsigned operation on an integer to detect overflow into the sign bit.

Packed decimal numbers are stored as unsigned byte quantities. The byte is treated as having one decimal digit in each half-byte (nibble); the digit in the high-order half-byte is the most significant. Hexadecimal values 0-9 are valid in each half-byte, and the range of a packed decimal number is 0-99. Addition and subtraction are performed in two steps. First an unsigned binary instruction is used to produce an intermediate result in register AL. Then an adjustment operation is performed which changes the intermediate value in AL to a final correct packed decimal result. Multiplication and division adjustments are not available for packed decimal numbers.

Unpacked decimal numbers are stored as unsigned byte quantities. The magnitude of the number is determined from the low-order half-byte; hexadecimal values 0-9 are valid and are interpreted as decimal numbers. The high-order half-byte must be zero for multiplication and division; it may contain any value for addition and subtraction. Arithmetic on unpacked decimal numbers is performed in two steps. The unsigned binary addition, subtraction, and multiplication operations are used to produce an intermediate result in register AL. An adjustment instruction then changes the value in AL to a final correct unpacked decimal number. Division is performed similarly, except that the adjustment is carried out on the numerator operand in register AL first, and then a following unsigned binary division instruction produces a correct result.

Unpacked decimal numbers are similar to the ASCII character representations of the digits 0-9. Note, however, that the high-order half-byte of an ASCII numeral is always 3H. Unpacked decimal arithmetic may be performed on ASCII numeric characters under the following conditions:

- The high-order half-byte of an ASCII numeral must be set to 0H prior to multiplication or division.
- Unpacked decimal arithmetic leaves the high-order half-byte set to 0H; it must be set to 3H to produce a valid ASCII numeral.

ARITHMETIC INSTRUCTIONS AND FLAGS

The 8086/8088 arithmetic instructions post certain characteristics of the result of the operation to six flags. Most of these flags can be tested by following the arithmetic instruction with a conditional jump instruction; the INTO (interrupt on overflow) instruction also may be used. The various instructions affect the flags differently, as explained in the instruction descriptions. However, they follow these general rules:

- CF (Carry Flag): If an addition results in a carry out of the high-order bit of the result, then CF is set; otherwise CF is cleared. If a subtraction results in a borrow into the high-order bit of the result, then CF is set; otherwise CF is cleared. Note that a signed carry is indicated by CF=OF. CF can be used to detect an unsigned overflow. Two instructions, ADC (add with carry) and SBB (subtract with borrow), incorporate the carry flag in their operations and can be used to perform multibyte (e.g., 32-bit, 64-bit) addition and subtraction.
- AF (Auxiliary Carry Flag): If an addition results in a carry out of the low-order half-byte of the result, then AF is set; otherwise AF is cleared. If a subtraction results in a borrow into the low-order half-byte of the result, then AF is set; otherwise AF is cleared. The auxiliary carry flag is provided for the decimal adjust instructions and ordinarily is not used for any other purpose.
- SF (Sign Flag): Arithmetic and logical instructions set the sign flag equal to the high-order bit (bit 7 or 15) of the result. For signed binary numbers, the sign flag will be 0 for positive results and I for negative results (so long as overflow does not occur). A conditional jump instruction can be used following addition or subtraction to alter the flow of the program depending on the sign of the result. Programs performing unsigned operations typically ignore SF since the high-order bit of the result is interpreted as a digit rather than a sign.
- ZF (Zero Flag): If the result of an arithmetic or logical operation is zero, then ZF is set; otherwise ZF is cleared. A conditional jump instruction can be used to alter the flow of the program if the result is or is not zero.
- PF (Parity Flag): If the low-order eight bits of an arithmetic or logical result contain an even number of 1-bits, then the parity flag is set; otherwise it is cleared.

  PF is provided for 8080/8085 compatibility; it also can be used to check ASCII characters for correct parity.
- OF (Overflow Flag): If the result of an operation is too large a positive number, or too small a negative number to fit in the destination operand (excluding the sign bit), then OF is set; otherwise OF is cleared. OF thus indicates signed arithmetic overflow; it can be tested with a conditional jump or the INTO (interrupt on overflow) instruction. OF may be ignored when performing unsigned arithmetic.

ADDITION

ADD *destination, source*

The sum of the two operands, which may be bytes or words, replaces the destination operand. Both operands may be signed or unsigned binary numbers (see AAA and DAA). ADD updates AF, CF, OF, PF, SF, and ZF.

ADC *destination, source*

ADC (Add with Carry) sums the operands, which may be bytes or words, adds one if CF is set, and replaces the destination operand with the result. Both operands may be signed or unsigned binary numbers (see AAA and DAA). ADC updates AF, CF, OF, PF, SF, and ZF. Since ADC incorporates a carry from a previous operation, it can be used to write routines to add numbers longer than 16 bits.

INC *destination*

INC (Increment) adds one to the destination operand. The operand may be a byte or a word and is treated as an unsigned binary number (see AAA and DAA). INC updates AF, OF, PF, SF, and ZF; it does not affect CF.

AAA

AAA (ASCII Adjust for Addition) changes the contents of register AL to a valid unpacked decimal number; the high-order half-byte is zeroed. AAA updates AF and CF; the content of OF, PF, SF, and ZF is undefined following execution of AAA.

DAA

DAA (Decimal Adjust for Addition) corrects the result of previously adding two valid packed decimal operands (the destination operand must have been register AL). DAA changes the content of AL to a pair of valid packed decimal digits. It updates AF, CF, PF, SF, and ZF; the content of OF is undefined following execution of DAA.

SUBTRACTION

SUB *destination, source*

The source operand is subtracted from the destination operand, and the result replaces the destination operand. The operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS). SUB updates AF, CF, OF, PF, SF, and ZF.

SBB *destination, source*

SBB (Subtract with Borrow) subtracts the source from the destination, subtracts one if CF is set, and returns the result to the destination operand. Both operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS). SBB updates AF, CF, OF, PF, SF, and ZF. Since it incorporates a borrow from a previous operation, SBB may be used to write routines that subtract numbers longer than 16 bits.

DEC *destination*

DEC (Decrement) subtracts one from the destination, which may be a byte or a word. DEC updates AF, OF, PF, SF, and ZF; it does not affect CF.

NEG *destination*

NEG (Negate) subtracts the destination operand, which may be a byte or a word, from 0 and returns the result to the destination. This forms the two's complement of the number, effectively reversing the sign of an interger. If the operand is zero, its sign is not changed. Attempting to negate a byte containing -128 or a word containing -32,768 causes no change to the operand and sets OF. NEG updates AF, CF, OF, PF, SF, and ZF. CF is always set except when the operand is zero, in which case it is cleared.

CMP *destination, source*

CMP (Compare) subtracts the source from the destination, which may be bytes or words, but does not return the result. The operands are unchanged, but the flags are updated and can be tested by a subsequent conditional jump instruction. CMP updates AF, CF, OF, PF, SF, and ZF. The comparison reflected in the flags is that of the destination to the source. If a CMP instruction is followed by a JG (Jump if Greater) instruction, for example, the jump is taken if the destination operand is greater than the source operand.

AAS

AAS (ASCII Adjust for Subtraction) corrects the result of a previous subtraction of two valid unpacked decimal operands (the destination operand must have been specified as register AL). AAS changes the content of AL to a valid unpacked decimal number; the high-order half-byte is zeroed. AAS updates AF and CF; the content of OF, PF, SF, and ZF is undefined following execution of AAS.

DAS

DAS (Decimal Adjust for Subtraction) corrects the result of a previous subtraction of two valid packed decimal operands (the destination operand must have been specified as register AL). DAS changes the content of AL to a pair of valid packed decimal digits. DAS updates AF, CF, PF, SF, and ZF; the content of OF is undefined following execution of DAS.

MULTIPLICATION

MUL *source*

MUL (Multiply) performs an unsigned multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length result is returned in AH and AL. If the source operand is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. The operands are treated as unsigned binary numbers (see AAM). If the upper half of the result (AH for byte source, DX for word source) is nonzero, CF and OF are set; otherwise they are cleared. When CF and OF are set, they indicate that AH or DX contains significant digits of the result. The content of AF, PF, SF, and ZF is undefined following execution of MUL.

IMUL *source*

IMUL (Integer Multiply) performs a signed multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length result is returned in AH and AL. If the source is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. If the upper half of the result (AH for byte source, DX for word source) is not the sign extension of the lower half of result, CF and OF are set; otherwise they are cleared. When CF and OF are set, they indicate that AH or DX contains significant digits of the result. The content of AF, PF, SF, and ZF is undefined following execution of IMUL.

AAM

AAM (ASCII Adjust for Multiply) corrects the result of a previous multiplication of two valid unpacked decimal operands. A valid 2-digit unpacked decimal number is derived from the content of AH and AL and is returned to AH and AL. The high-order half-bytes of the multiplied operands must have been 0H for AAM to produce a correct result. AAM updates PF, SF, and ZF; the content of AF, CF, and OF is undefined following execution AAM.

DIVISION

DIV *source*

DIV (divide) performs an unsigned division of accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the double-length dividend assumed to be in registers AL and AH. The single-length quotient is returned in AL, and the single-length remainder is returned in AH. If the source operand is a word, it is divided into the double-length dividend in registers AX and DX. The single-length quotient is returned in AX, and the single-length remainder is returned in DX. If the quotient exceeds the capacity of its destination register (FFH for byte source, FFFFFH for word source), as when division by zero is attempted, a type 0 interrupt is generated, and the quotient and remainder are undefined. Nonintegral quotients are truncated to integers. The content of AF, CF, OF, PF, SF, and ZF is undefined following execution of DIV.

IDIV *source*

IDIV (Integer Divide) performs a signed division of the accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the double-length dividend assumed to be in registers AL and AH; the single-length quotient is returned in AL, and the single-length remainder is returned in AH. For byte integer division, the maximum positive quotient is +127(7FH) and the minimum negative quotient is 127(81H). If the source operand is a word,it is divided into the double-length dividend in registers AX and DX; the single-length quotient is returned in AX, and the single-length remainder is returned in DX. For word integer division, the maximum positive quotient is +32,767 (7FFFH) and the minimum negative quotient is 32,767 (8001H). If the quotient is positive and exceeds the maximum, or is negative and is less than the minimum, the quotient and remainder are undefined, and a type 0 interrupt is generated. In particular, this occurs if division by 0 is attempted. Nonintegral quotients are truncated (toward 0) to integers, and the remainder has the same sign as the dividend. The content of AF, CF, OF, PF, SF, and ZF is undefined following IDIV.

AAD

AAD (ASCII Adjust for Division) modifies the numerator in AL before dividing two valid unpacked decimal operands so that the quotient produced by the division will be a valid unpacked decimal number. AH must be zero for the subsequent DIV to produce the correct result. The quotient is returned in AL, and the remainder is returned in AH; both high-order half-bytes are zeroed. AAD updates PF, SF, and ZF; the content of AF, CF, and OF is undefined following execution of AAD.

CBW

CBW (Convert Byte to Word) extends the sign of the byte in register AL throughout register AH. CBW does not affect any flags. CBW can be used to produce a double-length (word) dividend from a byte prior to performing byte division.

CWD

CWD (Convert Word to Doubleword) extends the sign of the word in register DX. CWD does not affect any flags. CWD can be used to produce a double-length (doubleword) dividend from a word prior to performing word division.

BIT MANIPULATION INSTRUCTIONS

The 8086 and 8088 provide three groups of instructions (Table A-4) for manipulating bits within both bytes and words: logical, shifts, and rotates.

Table A-4: Bit Manupulation Instructions

LOGICALS	
NOT	"Not" byte or word
AND	"And" byte or word
OR	"Inclusive or" byte or word
XOR	"Exclusive or" byte or word
TEST	"Test" byte or word
SHIFTS	
SHL/SAL	Shift logical/arithmetic left byte or word
SHR	Shift logical right byte or word
SAR	Shift arithmetic right byte or word
ROTATES	
ROL	Rotate left byte or word
ROR	Rotate right byte or word
RCL	Rotate through carry left byte or word
RCR	Rotate through carry right byte or word

LOGICAL

The logical instructions include the boolean operators "not," "and," "inclusive or", and "exclusive or", plus a TEST instruction that sets the flags, but does not alter either of its operands.

AND, OR, XOR and TEST affect the flags as follows: The overflow (OF) and carry (CF) flags are always cleared by logical instructions, and the content of the auxiliary carry (AF) flag is always undefined following execution of a logical instruction. The sign (SF), zero (ZF) and parity (PF) flags are always posted to reflect the result of the operation and can be tested by conditional jump instructions. The interpretation of these flags is the same as for arithmetic instructions. SF is set if the result is negative (high-order bit is 1), and is cleared if the result is positive (high-order bit is 0). ZF is set if the result is zero; it is otherwise cleared. PF is set if the result contains an even number of 1-bits (has even parity) and is cleared if the number of 1-bits is odd (the result has odd parity). Note that NOT has no effect on the flags.

NOT *destination*

NOT inverts the bits (forms the one's complement) of the byte or word operand.

AND *destination, source*

AND performs the logical "and" of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if both correspondence bits of the original operands are set; otherwise the bit is cleared.

OR *destination, source*

OR performs the logical "inclusive or" of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if either or both corresponding bits in the original operands are set; otherwise the result bit is cleared.

XOR *destination, source*

XOR (Exclusive Or) performs the logical "exclusive or" of the two operands and returns the result to the destination operand. A bit in the result is set if the corresponding bits of the original operands contain opposite values (one is set, the other is cleared); otherwise the result bit is cleared.

TEST *destination, source*

TEST performs the logical "and" of the two operands (byte or word), updates the flags, but does not return the result—i.e., neither operand is changed. If a TEST instruction is followed by a JNZ (Jump if Not Zero) instruction, the jump will be taken if there are any corresponding 1-bits in both operands.

SHIFTS

The bits in bytes and words may be shifted arithmetically or logically. Up to 255 shifts may be performed, according to the value of the count operand coded in the instruction. The count may be specified as the constant 1, or as register CL, allowing the shift count to be a variable supplied at execution time. Arithmetic shifts may be used to multiply and divide binary numbers by powers of two (see note in description of SAR). Logical shifts can be used to isolate bits in bytes or words.

Shift instructions affect the flags as follows: AF is always undefined following a shift operation. PF, SF, and ZF are updated normally, as in the logical instructions. CF always contains the value of the last bit shifted out of the destination operand. The content of OF is always undefined following a multibit shift. In a single-bit shift, OF is set if the value of the high-order (sign) bit was changed by the operation; if the sign bit retains its original value, OF is cleared.

SHL/SAL *destination, count*

SHL and SAL (Shift Logical Left and Shift Arithmetic Left) perform the same operation and are physically the same instruction. The destination byte or word is shifted left by the number of bits specified in the count operand. Zeros are shifted in on the right. If the sign bit retains its original value, then IF is cleared.

SHR *destination, source*

SHR (Shift Logical Right) shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Zeros are shifted in on the left. If the sign bit retains its original value, then OF is cleared.

SAR *destination, count*

SAR (Shift Arithmetic Right) shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Bits equal to the original high-order (sign) bit are shifted in on the left, preserving the sign of the original value. Note that SAR does not produce the same result as the dividend of an equivalent IDIV instruction if the destination operand is negative and 1-bits are shifted out. For example, shifting -5 right by one bit yields -3, while integer division of -5 by 2 yields -2. The difference in the instructions is that IDIV truncates all numbers toward zero, while SAR truncates positive numbers toward zero and negative numbers toward negative infinity.

ROTATES

Bits in bytes and words also may be rotated. Bits rotated out of an operand are not lost as in a shift, but are circled back into the other end of the operand. As in the shift instructions, the number of bits to be rotated is taken from the count operand, which may specify either a constant of 1, or the CL register. The carry flag may act as an extension of the operand in two of the rotate instructions, allowing a bit to be isolated in CF and then tested by a JC (Jump if Carry) or JNC (Jump if Not Carry) instruction.

Rotates affect only the carry and overflow flags. CF always contains the value of the last bit rotated out. On multibit rotates, the value of OF is always undefined. In single-bit rotates, OF is set if the operation changes the high-order (sign) bit of the destination operand. If the sign bit retains its original value, OF is cleared.

ROL *destination, count*

ROL (Rotate Left) rotates the destination byte or word left by the number of bits specified in the count operand.

ROR *destination, count*

ROR (Rotate Right) operates similar to ROL except that the bits in the destination byte or word are rotated right instead of left.

RCL *destination, count*

RCL (Rotate through Carry Left) rotates the bits in the byte or word destination operand to the left by the number of bits specified in the count operand. The carry flag (CF) is treated as "part of" the destination operand; that is, its value is rotated into the low-order bit of the destination, and is itself replaced by the high-order bit of the destination.

RCR *destination, count*

RCR (Rotate through Carry Right) operates exactly like RCL except that the bits are rotated right instead of left.

STRING INSTRUCTIONS

Five basic string operations, called primitives, allow strings of bytes or words to be operated on, one element (byte or word) at a time. Strings of up to 64k bytes may be manipulated with these instructions. Instructions are available to move, compare, and scan for a value, as well as for moving string elements to and from the accumulator (see Table A-5). These basic operations may be preceded by a special one-byte prefix that causes the instruction to be repeated by the hardware, allowing long strings to be processed much faster than would be possible with a software loop. The repetitions can be terminated by a variety of conditions, and a repeated operation may be interrupted and resumed.

Table A-5: String Instructions

REP	Repeat
REPE/REPZ	Repeat while equal/zero
REPNE/REPNZ	Repeat while not equal/not zero
MOVS	Move byte or word string
MOVSB/MOVSW	Move byte or word string
CMPS	Compare byte or word string
SCAS	Scan byte or word string
LODS	Load byte or word string
STOS	Store byte or word string

The string instructions operate quite similarly in many respects; the common characteristics are covered here and in Table A-6 and Figure A-2 rather than in the descriptions of the individual instructions. A string instruction may have a source operand, a destination operand, or both. The hardware assumes that a source string resides in the current data segment; a segment prefix byte may be used to override this assumption. A destination string must be in the current extra segment. The assembler checks the attributes of the operands to determine if the elements of the strings are bytes or words. The assembler does not, however, use the operand names to address the strings. Rather, the content of register SI (source index) is used as an offset to address the current element of the source string, and the content of register DI (destination index) is taken as the offset of the current destination string element. These registers must be initialized to point to the source/destination strings before executing the string instruction; the LDS, LES, and LEA instructions are useful in this regard.

Table A-6: String Instruction Register and Flag Use

SI	Index (offset) for source string
DI	Index (offset) for destination
CX	Repetition counter
AL/AX	Scan value Destination for LODS Source for STOS
DF	0=auto-increment SI, DI 1=auto-decrement SI, DI
ZF	Scan/compare terminator

The string instructions automatically update SI and/or DI in anticipation of processing the next string element. The DF (direction flag) setting determines whether the index registers are auto decremented (DF=1). If byte strings are being processed, SI and/or DI is adjusted by 1; the adjustment is 2 for word strings.

If a Repeat prefix has been coded, then register CX (count register) is decremented by 1 after each repetition of the string instruction; therefore, CX must be initialized to the number of repetitions desired before the string instruction is executed. If CX is 0, the string instruction is not executed, and control goes to the following instruction.

REP/REPE/REPZ/ REPNE/REPNZ

REP (Repeat), REPE (Repeat While Equal), REPZ (Repeat While Zero), REPNE (Repeat While Not Equal), and REPNZ (Repeat While Not Zero) are five mnemonics for two forms of the prefix byte that controls repetition of a subsequent string instruction. The different mnemonics are provided to improve program clarity. The repeat prefixes do not affect the flags.

REP is used in conjunction with the MOVS (Move String) and STOS (Store String) instructions and is interpreted as "repeat while not end-of-string" (CX not 0). REPE and REPZ operate identically and are physically the same prefix byte as REP. These instructions are used with the CMPS (Compare String) and SCAS (Scan String) instructions and require ZF (posted by these instructions) to be set before initiating the next repetition. REPNE and REPNZ are two mnemonics for the same prefix byte. These instructions function the same as REPE and REPZ, except that the zero flag must be cleared or the repetition is terminated. Note that ZF does not need to be initialized before executing the repeated string instruction.

Repeated string sequences are interruptable; the processor will recognize the interrupt before processing the next string element. System interrupt processing is not affected in any way. Upon return from the interrupt, the repeated operation is resumed from the point of interruption. Note, however, that execution does not resume properly if a second or third prefix (i.e., segment override or LOCK) has been specified in addition to any of the repeat prefixes. The processor "remembers" only one prefix in effect at the time of the interrupt—the prefix that immediately precedes the string instruction. After returning from the interrupt, processing resumes at this point, but any additional prefixes specified are not in effect. If more than one prefix must be used with a string instruction, interrupts may be disabled for the duration of the repeated execution. However, this will not prevent a nonmaskable interrupt from being recognized. Also, the time that the system is unable to respond to interrupts may be unacceptable if long strings are being processed.

MOVS *destination-string, source-string*

MOVS (Move String) transfers a byte or a word from the source string (addressed by SI) to the destination string (addressed by DI) and updates SI and DI to point to the next string element. When used in conjunction with REP, MOVS performs a memory-to-memory block transfer.

MOVSB/MOVSW

MOVSB and MOVSW are alternate mnemonics for the move string instruction. These mnemonics are coded without operands; they explicitly tell the assembler that a byte string (MOVSB) or a word string (MOVSW) is to be moved (when MOVS is coded, the assembler determines the string type from the attributes of the operands). These mnemonics are useful when the assembler cannot determine the attributes of a string—e.g., when a section of code is being moved.

CMPS *destination-string, source-string*

CMPS (Compare String) subtracts the destination byte or word (addressed by DI) from the source byte or word (addressed by SI). CMPS affects flags without altering either operand, updates SI and DI to point to the next string element, and updates AF, CF, OF, PF, SF, and ZF to reflect the relationship of the destination element to the source element. For example, if a JG (Jump if Greater) instruction follows CMPS, the jump is taken if the destination element is greater than the source element. If CMPS is prefixed with REPE or REPZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings are equal (ZF=1)." If CMPS is preceded by REPNE or REPNZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings are not equal (ZF=0)." Thus, CMPS can be used to find matching or differing string elements.

SCAS *destination-string*

SCAS (Scan String) subtracts the destination string element (byte or word) addressed by DI from the content of AL (byte string) or AX (word string) and updates the flags, but does not alter the destination string or the accumulator. SCAS also updates DI to point to the next string element and AF, CF, OF, PF, SF, and ZF to reflect the relationship of the scan value in AL/AX to the string element. If SCAS is prefixed with REPE or REPZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element scan value (ZF=1)." This form may be used to scan for departure from a given value. If SCAS is prefixed with REPNE or REPNZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element is not equal to scan-value (ZF=0)." This form may be used to locate a value in a string.

LODS *source-string*

LODS (Load String) transfers the byte or word string element addressed by SI to register AL or AX, and updates SI to point to the next element in the string. This instruction is not ordinarily repeated since the accumulator would be overwritten by each repetition, and only the last element would be retained. However, LODS is very useful in software loops as part of a more complex string function built up from string primitives and other instructions.

STOS *destination-string*

STOS (Store String) transfers a byte or word from register AL or AX to the string element addressed by DI and updates DI to point to the next location in the string. As a repeated operation, STOS provides a convenient way to initialize a string to a constant value (e.g., to blank out a print line).

PROGRAM TRANSFER INSTRUCTIONS

The sequence of execution of instructions in an 8086/8088 program is determined by the content of the code segment register (CS) and the instruction pointer (IP). The CS register contains the base address of the current code segment, the 64k portion of memory from which instructions are presently being fetched. The IP is used as an offset from the beginning of the code segment; the combination of CS and IP points to the memory location from which the next instruction is to be fetched. (Recall that under most operating conditions, the next instruction to be executed has already been fetched from memory and is waiting in the CPU instruction queue.) The program transfer

instructions operate on the instruction pointer and on the CS register; changing the content of these causes normal sequential execution to be altered. When a program transfer occurs, the queue no longer contains the correct instruction, and the BIU obtains the next instruction from memory using the new IP and CS values, passes the instruction directly to the EU, and then begins refilling the queue from the new location.

Four groups of program transfers are available in the 8086/8088: unconditional transfers, conditional transfers, iteration control instructions and interrupt-related instructions (see Table A-7). Only the interrupt-related instructions affect any CPU flags. As will be seen, however, the execution of many of the program transfer instructions is affected by the states of the flags.

Table A-7: Program Transfer Instructions

UNCONDITIONAL TRANSFERS	
CALL	Call procedure
RET	Return from procedure
JMP	Jump
CONDITIONAL TRANSFERS	
JA/JNBE	Jump if above/not below or equal
JAE/JNB	Jump if above or equal/not below
JB/JNAE	Jump if below/not above or equal
JBE/JNA	Jump if below or equal/not above
JC	Jump if carry
JE/JZ	Jump if equal/zero
JG/JNLE	Jump if greater/not less or equal
JGE/JNL	Jump if greater or equal/not less
JL/JNGE	Jump if less/not greater or equal
JLE/JNG	Jump if less or equal/not greater
JNC	Jump if not carry
JNE/JNZ	Jump if not equal/not zero
JNO	Jump if not overflow
JNP/JPO	Jump if not parity/parity odd
JNS	Jump if not sign
JO	Jump if overflow
JP/JPE	Jump if parity/parity even
JS	Jump if sign
ITERATION CONTROLS	
LOOP	Loop
LOOPE/LOOPZ	Loop if equal/zero
LOOPNE/LOOPNZ	Loop if not equal/not zero
JCXZ	Jump if register CX=0
INTERRUPTS	
INT	Interrupt
INTO	Interrupt if overflow
IRET	Interrupt return

UNCONDITIONAL TRANSFERS

The unconditional transfer instructions may transfer control to a target instruction within the current code segment (intrasegment transfer) or to a different code segment (intersegment transfer). The ASM-86 assembler terms an intrasegment target NEAR and an intersegment target FAR. The transfer is made unconditionally any time the instruction is executed.

CALL ***procedure-name***

CALL activates an out-of-line procedure, saving information on the stack to permit a RET (return) instruction in the procedure to transfer control back to the instruction following the CALL. The assembler generates one of two types of CALL instruction; the type depends on whether the programmer has defined the procedure name as NEAR or FAR. For control to return properly, the type of CALL instruction must match the type of RET instruction that exits from the procedure. (The potential for a mismatch exists if the procedure and the CALL are contained in separately assembled programs.) Different forms of the CALL instruction allow the address of the target procedure to be obtained from the instruction itself (direct CALL) or from a memory location or register referenced by the instruction (indirect CALL). In the following descriptions, bear in mind that the processor automatically adjusts IP to point to the next instruction to be executed before saving it on the stack.

For an intrasegment direct CALL, SP (the stack pointer) is decremented by two and IP is pushed onto the stack. The relative displacement (up to +32k) of the target procedure from the CALL instruction is then added to the instruction pointer. This form of the CALL instruction is self-relative and is appropriate for position-independent (dynamically relocatable) routines in which the CALL and its target are in the same segment and are moved together.

An intrasegment indirect CALL may be made through memory or through a register. SP is decremented by two and IP is pushed onto the stack. The offset of the target procedure is obtained from the memory word or 16-bit general register referenced in the instruction and replaces IP.

For an intersegment direct CALL, SP is decremented by two, and CS is pushed onto the stack. CS is replaced by the segment word contained in the instruction. SP again is decremented by two. IP is pushed onto the stack and is replaced by the offset word contained in the instruction.

For an intersegment indirect CALL (which only may be made through memory), SP is decremented by two, and CS is pushed onto the stack. CS is then replaced by the content of the second word of the doubleword memory pointer referenced by the instruction. SP again is decremented by two, and IP is pushed onto the stack and is replaced by the content of the first word of the doubleword pointer referenced by the instruction.

RET
optional-pop-value

RET (Return) transfers control from a procedure back to the instruction following the CALL that activated the procedure. The assembler generates either an intrasegment RET, if the programmer has defined the procedure NEAR, or an intersegment RET, if the procedure has been defined as FAR. RET pops the word at the top of the stack (pointed to by register SP) into the instruction pointer and increments SP by two. If RET is intersegment, the word at the new top of stack is popped into the CS register, and SP is again incremented by two. If an optional pop value has been specified, RET adds that value to SP. This feature may be used to discard parameters pushed onto the stack before the execution of the CALL instruction.

JMP *Target*

JMP unconditionally transfers control to the target location. Unlike a CALL instruction, JMP does not save any information on the stack, and no return to the instruction following the JMP is expected. Like CALL, the address of the target operand may be obtained from the instruction itself (direct JMP) or from memory or a register referenced by the instruction (indirect JMP).

An intrasegment direct JMP changes the instruction pointer by adding the relative displacement of the target from the JMP instruction. If the assembler can determine that the target is within 127 bytes of the JMP, it automatically generates a two-byte form of this instruction called a SHORT JMP; otherwise, it generates a NEAR JMP that can address a target within +32k. Intrasegment direct JMPS are self-relative and are appropriate in position-independent (dynamically relocatable) routines in which the JMP and its target are in the same segment and are moved together.

An intrasegment indirect JMP may be made either through memory or through a 16-bit general register. In the first case, the content of the word referenced by the instruction replaces the instruction pointer. In the second case, the new IP value is taken from the register named in the instruction.

An intersegment direct JMP replaces IP and CS with values contained in the instruction.

An intersegment indirect JMP may be made only through memory. The first word of the doubleword pointer referenced by the instruction replaces IP, and the second word replaces CS.

CONDITIONAL TRANSFERS

The conditional transfer instructions are jumps that may or may not transfer control depending on the state of the CPU flags at the time the instruction is executed. These 18 instructions (see Table A-8) each test a different combination of flags for a condition. If the condition is true, then control is transferred to the target specified in the instruction. If the condition is false, then control passes to the instruction that follows the conditional jump. All conditional jumps are SHORT, that is, the target must be in the current code segment and within -128 to +127 bytes of the first byte of the next instruction (JMP 00H jumps to the first byte of the next instruction). Since the jump is made by adding the relative displacement of the target to the instruction pointer, all conditional jumps are self-relative and are appropriate for position-independent routines.

Table A-8: Interpretation of Conditional Transfers

MNEMONIC	CONDITION TESTED	"JUMP IF. . ."
JA/JNBE	(CF or ZF)=0	above/not below or equal
JAE/JNB	CF=0	above or equal/not below
JB/JNAE	CF=1	below/not above or equal
JBE/JNA	(CF or ZF)=1	below or equal/not above
JC	CF=1	carry
JE/JZ	ZF=1	equal/zero
JG/JNLE	((SF xor OF) or ZF)=0	greater/not less or equal
JGE/JNL	(SF xor OF)=0	greater or equal/not less
JL/JNGE	(SF xor OF)=1	less/not greater or equal
JLE/JNG	((SF xor OF) or ZF)=1	less or equal/not greater
JNC	CF=0	not carry
JNE/JNZ	ZF=0	not equal/not zero
JNO	OF=0	not overflow
JNP/JPO	PF=0	not parity/parity odd
JNS	SF=0	not sign
JO	OF=1	overflow
JP/JPE	PF=1	parity/parity equal
JS	SF=1	sign

NOTE: "above" and "below" refer to the relationship of two unsigned values; "greater" and "less" refer to the relationship of two signed values.

ITERATION CONTROL

The iteration control instructions can be used to regulate the repetition of software loops. These instructions use the CX register as a counter. Like the conditional transfers, the iteration control instructions are self-relative and may only transfer to targets that are within -128 to +127 bytes of themselves, i.e., they are SHORT transfers.

LOOP *short-label*

LOOP decrements CX by 1 and transfers control to the target operand if CX is not 0; otherwise the instruction following LOOP is executed.

LOOPE/LOOPZ *short-label*

LOOPE and LOOPZ (Loop While Equal and Loop While Zero) are different mnemonics for the same instruction (similar to the REPE and REPZ repeat prefixes). CX is decremented by 1, and control is transferred to the target operand if CX is not 0 and if ZF is set; otherwise the instruction following LOOPE or LOOPZ is executed.

LOOPNE/LOOPNZ *short-label*

LOOPNE and LOOPNZ (Loop While Not Equal and Loop While Not Zero) are also synonyms for the same instruction. CX is decremented by 1, and control is transferred to the target operand if CX is not 0 and ZF is clear; otherwise the next sequential instruction is executed.

JCXZ *short-label*

JCXZ (Jump If CX Zero) transfers control to the target operand if CX is 0. This instruction is useful at the beginning of a loop to bypass the loop if CX has a zero value, i.e., to execute the loop zero times.

INTERRUPT INSTRUCTIONS

The interrupt instructions allow interrupt service routines to be activated by programs as well as by external hardware devices. The effect of software interrupts is similar to hardware-initiated interrupts. However, the processor does not execute an interrupt acknowledge bus cycle if the interrupt originates in software or with an NMI. The effect of the interrupt instructions on the flags is covered in the description of each instruction.

INT *Interrupt-type*

INT (Interrupt) activates the interrupt procedure specified by the interrupt-type operand. INT decrements the stack pointer by two, pushes the flags onto the stack, and clears the trap flag (TF) and interrupt-enable flag (IF) to disable single-step and maskable interrupts. The flags are stored in the format used by the PUSHF instruction. SP is decremented again by two, and the CS register is pushed onto the stack. The address of the interrupt pointer is calculated by multiplying interrupt-type by four; the second word on the interrupt pointer replaces CS. SP again is decremented by two, and IP is pushed onto the stack and is replaced by the first word of the interrupt pointer. If interrupt-type=3, the assembler generates a short (1 byte) form of the instruction, known as the breakpoint interrupt.

Software interrupts can be used as supervisor calls—requests for service from an operating system. A different interrupt-type can be used for each type of service that the operating system could supply for an application program. Software interrupts also may be used to check out interrupt service procedures written for hardware-initiated interrupts.

INTO

INTO (Interrupt on Overflow) generates a software interrupt if the overflow flag (OF) is set; otherwise control proceeds to the following instruction without activating an interrupt procedure. INTO addresses the target interrupt pointer at location 10H; it clears the TF and IF flags and otherwise operates like INT. INTO may be written following an arithmetic or logical operation to activate an interrupt procedure if overflow occurs.

IRET

IRET (Interrupt Return) transfers control back to the point of interruption by popping IP, CS, and the flags from the stack. IRET thus affects all flags by restoring them to previously saved values. IRET is used to exit any interrupt procedure, whether activated by hardware or software.

PROCESSOR CONTROL INSTRUCTIONS

These instructions (see Table A-9) allow programs to control various CPU functions. One group of instructions updates flags, and another group is used primarily for synchronizing the 8086 or 8088 with external events. A final instruction causes the CPU to do nothing. Except for the flag operations, none of the processor control instructions affect the flags.

Table A-9: Processor Control Instructions

FLAG OPERATIONS	
STC	Set carry flag
CLC	Clear carry flag
CMC	Complement carry flag
STD	Set direction flag
CLD	Clear direction flag
STI	Set interrupt-enable flag
CLI	Clear interrupt-enable flag
EXTERNAL SYNCHRONIZATION	
HLT	Halt until interrupt or reset
WAIT	Wait for $\overline{\text{TEST}}$ pin active
ESC	Escape to external processor
LOCK	Lock bus during next instruction
NO OPERATION	
NOP	No operation

FLAG OPERATIONS

CLC CLC (Clear Carry flag) zeroes the carry flag (CF) and affects no other flags. It (and CMC and STC) is useful in conjunction with the RCL and RCR instructions.

CMC CMC (Complement Carry flag) toggles CF to its opposite state and affects no other flags.

STC STC (Set Carry flag) sets CF to 1 and affects no other flags.

CLD CLD (Clear Direction flag) zeroes DF, causing the string instructions to auto-increment the SI and/or DI index registers. CLD does not affect any other flags.

STD STD (Set Direction flag) sets DF to 1, causing the string instructions to autodecrement the SI and/or DI index registers. STD does not affect any other flags.

CLI CLI (Clear Interrupt-enable flag) zeroes IF. When the interrupt-enable flag is cleared, the 8086 and 8088 do not recognize an external interrupt request that appears on the INTR line; in other words, maskable interrupts are disabled. A nonmaskable interrupt appearing on the NMI line, however, is honored, as is a software interrupt. CLI does not affect any other flags.

STI STI (Set Interrupt-enable flag) sets IF to 1, enabling processor recognition of maskable interrupt requests appearing on the INTR line. Note however, that a pending interrupt will not actually be recognized until the instruction following STI has executed. STI does not affect any other flags.

EXTERNAL SYNCHRONIZATION

HLT

HLT (Halt) causes the 8086/8088 to enter the halt state. The processor leaves the halt state upon activation of the RESET line, upon receipt of a nonmaskable interrupt request on NMI or, if interrupts are enabled, upon receipt of a maskable interrupt request on INTR. HLT does not affect any flags. It may be used as an alternative to an endless software loop in situations where a program must wait for an interrupt.

WAIT

WAIT causes the CPU to enter the wait state while its TEST line is not active. WAIT does not affect any flags.

ESC *external-opcode, source*

ESC (Escape) provides a means for an external processor to obtain an opcode and possibly a memory operand from the 8086 or 8088. The external opcode is a 6-bit immediate constant that the assembler encodes in the machine instruction it builds (see Table A-10). An external processor may monitor the system bus and capture this opcode when the ESC is fetched. If the source operand is a register, the processor does nothing. If the source operand is a memory variable, the processor obtains the operand from memory and discards it. An external processor may capture the memory operand when the processor reads it from memory.

LOCK

LOCK is a 1-byte prefix that causes the 8086/8088 (configured in maximum mode) to assert its bus LOCK signal while the following instruction executes. LOCK does not affect any flags.

NO OPERATION:NOP

NOP (No Operation) causes the CPU to do nothing. NOP does not affect any flags.

INSTRUCTION SET REFERENCE INFORMATION

Appendix I provides detailed operational information for the 8086/8088 instruction set.

Appendix B EXPANSION BUS DEFINITION

The Expansion Bus is basically a buffered extension of the systems 8088 processor plus additional control and timing signals required to interface the system. The expansion bus consists of—

- A multiplexed buffered data bus, BD0-BD7
- A buffered address bus, A8-A19
- Various timing, control, interrupt, and power lines

Table B-1: Expansion Bus Pin Definition

PIN	SIGNAL	I/O	DESCRIPTION
50	A19	IO	Buffered Address Bits 8 to 19: These lines are driven from the 8088 during normal operation and are valid from the falling edge of ALE to the rising edge of the next ALE. If an external device takes control of the system via HOLD and HOLD ACKNOWLEDGE, these lines are tri-stated.
1	A18	IO	
49	A17	IO	
2	A16	IO	
48	A15	IO	
3	A14	IO	
47	A13	IO	
4	A12	IO	
46	A11	IO	
5	A10	IO	
45	A9	IO	
6	A8	IO	
29	BD7	IO	Time Multiplexed Buffered Address/Data Bus: During normal operation, the lower 8 bits of address, AD0-AD7, are valid on the falling edge of ALE.
22	BD6	IO	
28	BD5	IO	
23	BD4	IO	
27	BD3	IO	
24	BD2	IO	
26	BD1	IO	
25	BDO	IO	
9	ALE	O	Buffered Address Latch Enable: Processor signal which indidates BD0-BD7 contain valid addresses. Typically used to latch low-order 8 bits of address.
11	RD	O	Buffered Read Strobe: Processor signal indicating a read cycle.
14	$\overline{WR}$	O	Buffered Write Strobe: Processor signal indicating a write cycle.
8	$\overline{DEN}$	O	Buffered Data Enable: Provided by the processor for use as an enable for transceivers.
33	$\overline{DLATCH}$	O	Data Latch: The falling edge of this signal may be used to strobe data generated from a processor read access.
30	$\overline{EXTIO}$	I	External IO: Control line which prevents internal data bus buffers from conflicting with external buffers when mapping external IO into address space E0000 to EFFFF. CSEN should be used as a control signal to disable internal buffers via EXTIO and enable external buffers if using address space E0000 to EFFFF. Addresses used by the system cannot be disabled by EXTIO.

19	CSEN	O	Chip Select Enable: This line is synchronized to PHASE2. It is true from a falling edge of PHASE2 to the next falling edge of PHASE2, when address space E0000 to EFFFF is accessed.
40	CLK15B	O	15-Mhz Clock: Signal from which all system timing is derived. Its period is 66.6 nanoseconds with a 50%±10% duty cycle.
38	CLK5	O	5-Mhz Clock: Signal is in phase with the 8088 clock input. Its period is 200 nanoseconds with a 33% duty cycle.
20	PHASE2	O	1-Mhz Clock: Signal is asynchronous with CLK5. Its period is 1 microsecond with a 40/60% duty cycle. Useful to interface 6800-type I/O circuits.
21	XACK	I	External Acknowledge: This line is normally high and may be pulled low by external devices resulting in pulling the 8088 Ready input low, generating wait states. This line is resynchronized by the system logic.
17	$\overline{\text{HOLD}}$	I	Input to the 8088. This is an external request for control of the system buses.

Table B-1: Expansion Bus Pin Definition (Concluded)

PIN	SIGNAL	I/O	DESCRIPTION
18	HLDA	O	Buffered Hold Acknowledge: System response to "HOLD" request. When true (high) the following signals are tri-stated: A8-A19 BD0-BD7 ALE IO/$\overline{\text{M}}$ $\overline{\text{RD}}$ $\overline{\text{WR}}$ DT/$\overline{\text{R}}$ $\overline{\text{DEN}}$ $\overline{\text{SSO}}$ $\overline{\text{INTA}}$ $\overline{\text{DLATCH}}$ is controlled by external logic.
41	READY	O	Status Line: This line reflects the synchronized "ready" input to the 8088.
10	IO/$\overline{\text{M}}$	O	Buffered 8088 Status Line: Distinguishes between a memory or I/O bus cycle.
7	$\overline{\text{SSO}}$	O	Buffered 8088 Status Line.
12	DT/$\overline{\text{R}}$	O	Buffered Data Transmit/Receive: Processor signal typically used to control the direction of system transceivers. The combination of IO/$\overline{\text{M}}$, DT/$\overline{\text{R}}$, and $\overline{\text{SSO}}$ provide current bus cycle status:

IO/$\overline{M}$	DT/$\overline{R}$	$\overline{SSO}$	DESCRIPTION
0	0	0	Instruction fetch
0	0	1	Read from memory
0	1	0	Write from memory
0	1	1	Passive (no bus cycle)
1	0	0	Interrupt acknowledge
1	0	1	Read from I/O
1	1	0	Write to I/O
1	1	1	Halt

15	$\overline{NMI}$	I	Non-Maskable Interrupt: An edge-triggered input which causes a type-2 interrupt. A transition from high to low initiates the interrupt at the end of the current instruction.
16	$\overline{IRQ}$	I	Interrupt Request: This input should be driven with an open collector driver; it is "collector ORed" with five 6522s and one 6852 and is pulled to +5 volts through a 3.3K ohm resistor. A low level on any of these circuits generates a high level input to the system 8259 at IR3 level.
43	IR4	I	Interrupt Request Level 4: Direct access to IR4 of the system 8259.
42	IR5	I	Interrupt Request Level 5: Direct access to IR5 of the system 8259.
13	RESET	O	System Reset: Generated at power on or from the Reset switch.

PIN	SIGNAL	DESCRIPTION
44	Ground	
39	Ground	
35	Ground	
31	Ground	
37	+5volts	
36	+5volts	250 ma/expansion board
34	+12 volts	250 ma/expansion board
32	-12 volts	50 ma/expansion board

Table B-2: Expansion Bus Loading

SIGNAL	NORMAL USAGE I/O	INTERNAL LOAD	EXTERNAL DRIVE
Tri-Stated Lines			
A8-19	O	4	4
BDO-7	IO	5	4
ALE	O	5	4
RD	O	4	4
WR	O	4	4
DEN	O	4	4
IO/M	O	2	4
SSO	O	1	4
DT/R	O	4	4
TTL Outputs			
DLATCH	O	-	4*
CSEN	O	-	4*
C1K15B	O	-	1*
C1K5	O	-	4*
PHASE2	O	-	1*
HLDA	O	-	1*
READY	O	-	4
RESET	O	-	4

NOTE: All loads are 74LSXX loads of .4ma. External drive, as specified, is for each of the four slots available. Care must be taken to ensure adequate drive for other expansion modules which may be installed in the system.

*If required, buffer through one common IC package, such as 74LSO4.

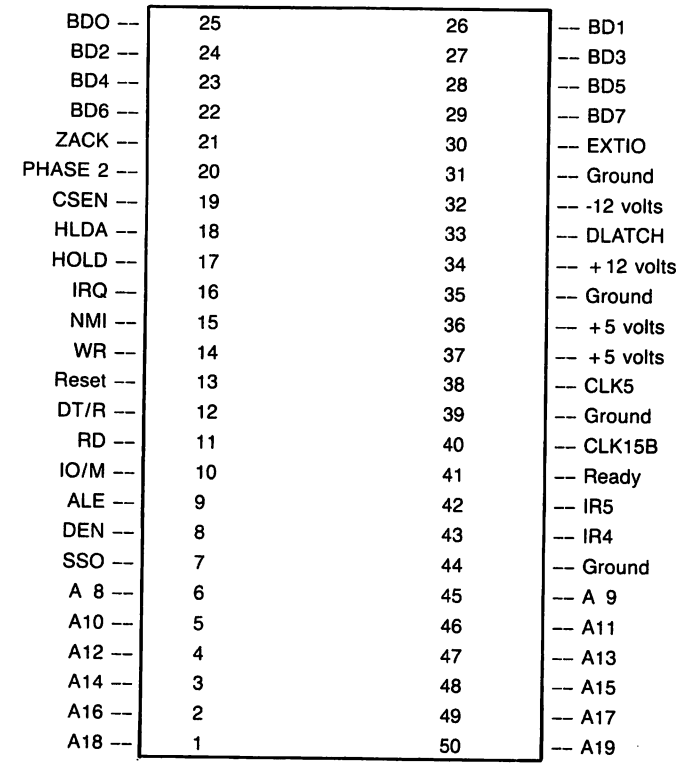
Table B-3: Inputs Driven with Open Collector Drivers

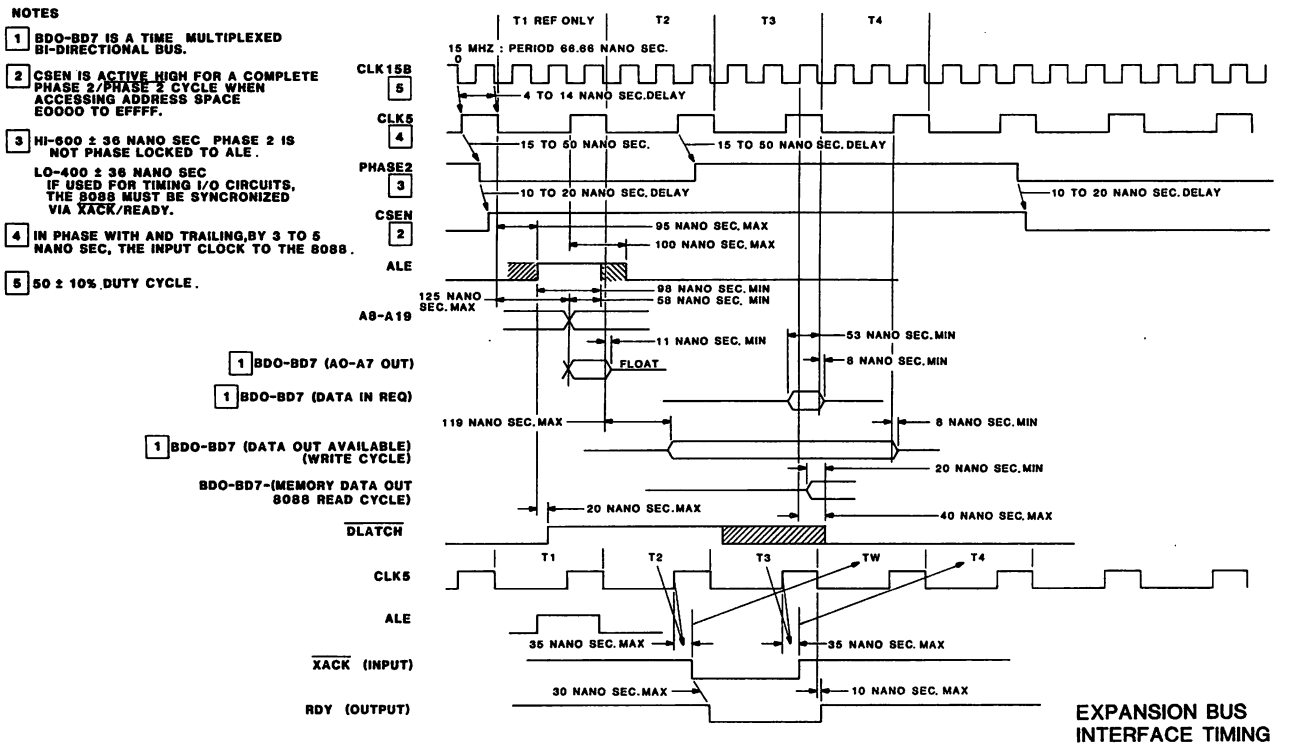
SIGNAL	INTERNAL LOAD	PULLUP PROVIDED
$\overline{\text{EXTIO}}$	2	2.2K
$\overline{\text{XACK}}$	1	2.2K
$\overline{\text{HOLD}}$	1	2.2K
$\overline{\text{NMI}}$	1	2.2K
$\overline{\text{IRQ}}$	1	3.3K

Table B-4: Inputs Direct to System 8259

IR4
IR5

Figure B-1: Expansion Connector





EXPANSION BUS INTERFACE TIMING

Figure B-2: Expansion Bus Interface Timing

Appendix C MEMORY MAPPED I/O ADDRESS AND BIT ASSIGNMENTS

Table C-1: 8259A (PIC IOD0) Address: E0000-E0001

INTERRUPT LEVEL	SIGNAL NAME	DESCRIPTION
IR0	SYN	SYNC DETECT
IR1	COMM	SERIAL COMMUNICATIONS (7201)
IR2	TIMER	8253 TIMER
IR3	PARALLEL	ALL 6522 IRQ (INCLUDING DISK)
IR4	IR4	EXPANSION IR4
IR5	IR5	EXPANSION IR5
IR6	KBINT	KEYBOARD DATA READY
IR7	VINT	VERTICAL SYNC OR NONSPECIFIC INTERRUPT

Table C-2: 8253 (TIMER-IOD1) Address: E0020-E0023

I/O NAME	SIGNAL NAME	DESCRIPTION
CLK2	100KHZ	CLOCK INPUT (FOR TIME OF DAY)
GATE2	+5 V	
OUT2	TIMER	INTERRUPT FOR TIME OF DAY
GLK1	1.25 MHZ	CLOCK INPUT FOR SERIAL PORT B
GATE1	+5 V	
OUT1	MUX SERIAL B	TO SERIAL PORT B MUX
CLK0	1.25 MHZ	CLOCK INPUT FOR SERIAL PORT A
GATE0	+5 V	
OUT0	MUX SERIAL A	TO SERIAL PORT A MUX

Table C-3: 7201(COMM.CTLR IOD2)
Address: E0040-E0043

I/O NAME	SIGNAL NAME	DESCRIPTION
RXCA	J8-17	RECEIVE CLK A
TXCA	J8-15	TRANSMIT CLK A
RXDA	J8-3	RECEIVE DATA A
TXDA	J8-2	TRANSMIT DATA A
CTSA	J8-5	CLEAR TO SEND A
RTSA	J8-4	REQUEST TO SEND A
DCDA	J8-8	DATA CARRIER DETECT A INPUT
DTRA	J8-20	DATA TERMINAL READY A
RXCB	J9-17	RECEIVE CLK B
TXCB	J9-15	TRANSMIT CLK B
RXDB	J9-3	RECEIVE DATA B
TXDB	J9-2	TRANSMIT DATA B
CTSB	J9-5	CLEAR TO SEND B
RTSB	J9-4	REQUEST TO SEND B
DCDB	J9-8	DATA CARRIER DETECT B INPUT
DTRB	J9-20	DATA TERMINAL READY B

Table C-4: HD46505S (CRTC CSO)
Address: E8000-E8001

INTERRUPT LEVEL	SIGNAL NAME	DESCRIPTION
MA13	HIRES	HIRES ENABLE OUTPUT
MA12	DOT ADDR	32K WORD PAGE SELECT OUTPUT (1=UPPER)

Table C-5: 6522 (VIA 1 CS1)
Address: E8020-E802F

I/O NAME	SIGNAL NAME	DESCRIPTION
PA0	DIO1	Parallel data bit 0, IN/OUT
PA1	DIO2	Parallel data bit 1, IN/OUT
PA2	DIO3	Parallel data bit 2, IN/OUT
PA3	DIO4	Parallel data bit 3, IN/OUT
PA4	DIO5	Parallel data bit 4, IN/OUT
PA5	DIO6	Parallel data bit 5, IN/OUT
PA6	DIO7	Parallel data bit 6, IN/OUT
PA7	DIO8	Parallel data bit 7, IN/OUT
CA1	NRFD	Parallel NRFD interrupt input
CA2	NDAC	Parallel NDAC interrupt input
PB0	DAV	Parallel DAV, IN/OUT
PB1	EOI	Parallel EOI, IN/OUT
PB2	REN	Parallel REN, IN/OUT
PB3	ATN	Parallel ATN, IN/OUT
PB4	IFC	Parallel IFC, IN/OUT
PB5	SRQ	Parallel SRQ, IN/OUT
PB6	NRFD	Parallel NRFD, IN/OUT
PB7	NDAC	Parallel NDAC, IN/OUT
CB1	N.C.	
CB2	CODEC VOL	Pulse width control CODEC Vol output (TZ)

Table C-6: 6522 (VIA 2 CS2)
Address: E8040-E804F

I/O NAME	SIGNAL NAME	DESCRIPTION
PA0	$\overline{\text{INT}}$/EXTA	Serial A clock select (LOW=INT)
PA1	$\overline{\text{INT}}$/EXTB	Serial B clock select (LOW=INT)
PA2	RIA	Serial A ring indicate (J8-22)
PA3	DSRA	Serial A data set ready (J8-6)
PA4	RIB	Serial B ring indicate (J9-22)
PA5	DSRB	Serial B data set ready (J9-6)
PA6	KBDATA	Data from keyboard
PA7	$\overline{\text{VERT}}$	Vertical signal input (from CRTC)
CA1	NC	
CA2	SRQ/BUSY	Parallel port IN/OUT
PB0	TALK/LISTEN	Parallel port direction, control, output
PB1	KBACKCTL	Keyboard acknowledge, control, output
PB2	BRT0	LSB of brightness control, output
PB3	BRT1	Intermediate bit of brightness control, output
PB4	BRT2	MSB of brightness control, output
PB5	CONT0	LSB of contrast control, output
PB6	$\overline{\text{CONT1}}$	Intermediate bit of contrast control, output
PB7	$\overline{\text{CONT2}}$	MSB of contrast control, output
CB1	KBRDY	Key data ready, input
CB2	KBDATA	Shift register input

Table C-7: 6852 (SSDA CS3)
Address: E8060-E806F

I/O NAME	SIGNAL NAME	DESCRIPTION
RXCLK		Inverted input from PB7 of VIA3 (CODEC CLOCK)
TXCLK		Inverted input from PB7 of VIA3 (CODEC CLOCK)
RXDATA		Input digital data from CODEC
TXDATA		Digital data output to CODEC
SM/DTR		Encode/Decode control for CODEC (Low=Decode, or transmit)
DCD		Inverted input from SM/DTR of this chip
CTS		Input from SM/DTR of this chip

Table C-8: 6522 (VIA 3 CS4)
Address: E8080-E808

I/O NAME	SIGNAL NAME	DESCRIPTION
PA0	J5-16	Control Port
PA1	J5-18	Control Port
PA2	J5-20	Control Port
PA3	J5-22	Control Port
PA4	J5-24	Control Port
PA5	J5-26	Control Port
PA6	J5-28	Control Port
PA7	J5-30	Control Port
CA1	J5-12	Control Port
CA2	J5-14	Control Port
PB0	J5-32	Control Port
PB1	J5-34	Control Port
PB2	J5-36	Control Port
PB3	J5-38	Control Port
PB4	J5-40	Control Port
PB5	J5-42	Control Port
PB6	J5-44	Control Port
PB7	J5-46	CODEC Clock Output
CB1	J5-48	Control Port
CB2	J5-50	Control Port

Table C-9: 6522 (VIA 4 CS5)
Address: E80A0-E80AF

I/O NAME	SIGNAL NAME	DESCRIPTION
PA0	L0MS0	Drive 0 motor speed, outputs (also used as a data bus to load 8048 parameters during motor speed controller initialization)
PA1	L0MS1	
PA2	L0MS2	
PA3	L0MS3	
PA4	ST0A	Drive 0 stepper phase, outputs
PA5	ST0B	
PA6	ST0C	
PA7	ST0D	
CA1	DS0	Door 0 sense interrupt, input
CA2	MODE	Write sync
PB0	L1MS0	Drive 1 motor speed, outputs
PB1	L1MS1	
PB2	L1MS2	
PB3	L1MS3	
PB4	ST1A	Drive 1 stepper phase, outputs
PB5	ST1B	
PB6	ST1C	
PB7	ST1D	
CB1	DS1	Door 1 sense interrupt, input
CB2	N.C.	

Table C-10: 6522 (VIA 6 CS6)
Address: E80C0-E80CF

I/O NAME	SIGNAL NAME	DESCRIPTION
PA0	LED0A	LED, drive A, output
PA1	TRK0D0	Track 0, drive A sense, input
PA2	LED1A	LED, drive B, output
PA3	TRK0D1	Track 0, drive B sense, input
PA4	Side Select	Dual side select, output
PA5	Drive Select	Select drive A/B, output
PA6	WPS	Write protect sense, input
PA7	SYNC	Disk sync detect, input
CA1	GCRERR	GCR error input
CA2	DRW	Disk read/write CTRL, output
*PB0	RDY0	Motor speed status, drive A
*PB1	RDY1	Motor speed status, drive B
PB2	SCRESET	Motor speed controller (8048) reset, output
PB3	DS1	Door B sense, input
PB4	DS0	Door A sense, input
PB5		Single/Double sided
PB6		Stepper enable A
PB7		Stepper enable B
CB1	N.C.	
CB2	Erase	Erase head On/Off, output

*Also used as handshake lines during speed controller initialization.

Table C-11: 6522 (VIA 5 CS7)
Address: E80E0-E80EF

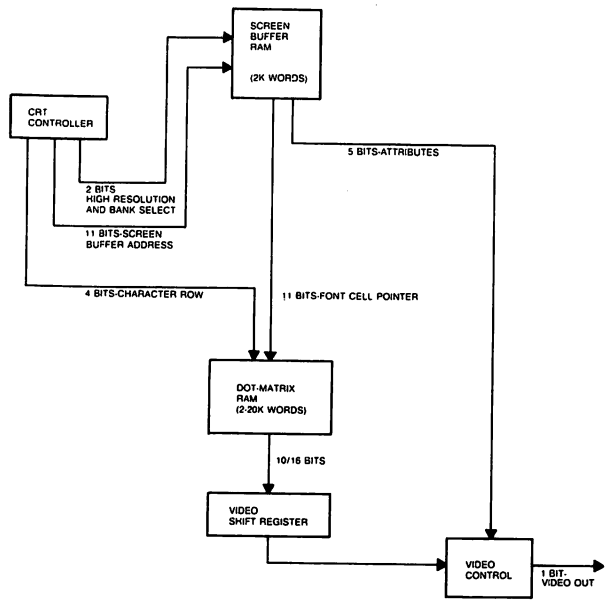
I/O NAME	SIGNAL NAME	DESCRIPTION
PA0	E0	
PA1	E1	
PA2	I2	
PA3	E2	Disk data inputs
PA4	E4	
PA5	E5	
PA6	I7	
PA7	E6	
CA1	BRDY	Byte ready input
CA2	RDY0	Motor speed status interrupt, drive 0
PB0	WD0	
PB1	WD1	
PB2	WD2	
PB3	WD3	Disk data outputs
PB4	WD4	
PB5	WD5	
PB6	WD6	
PB7	WD7	
CB1	N.C.	
CB2	RDY1	Motor speed status interrupt, drive 1

Appendix D THE DISPLAY SYSTEM

INTRODUCTION

The display hardware is a memory-mapped raster scan system. The display RAM physically occupies 4K bytes, starting at F0000H, plus from 4K to 40K bytes of the lower 128 bytes in the 8088 memory map. The display RAM is organized in two separate banks, which operate in a pipelined fashion (see Figure D-1). The first bank is the screen bufgfer; it contains the characters which are to be displayed on the screen. The screen buffer also contains attribute information for each character location. The character selection code (called the font cell pointer), together with the character row number (0-15) is used as the address for the second bank, which contains patterns for the characters (font cells). To generate video, the font cell patterns are accessed and latched into the video shift register.

Figure D-1: Display System Organization

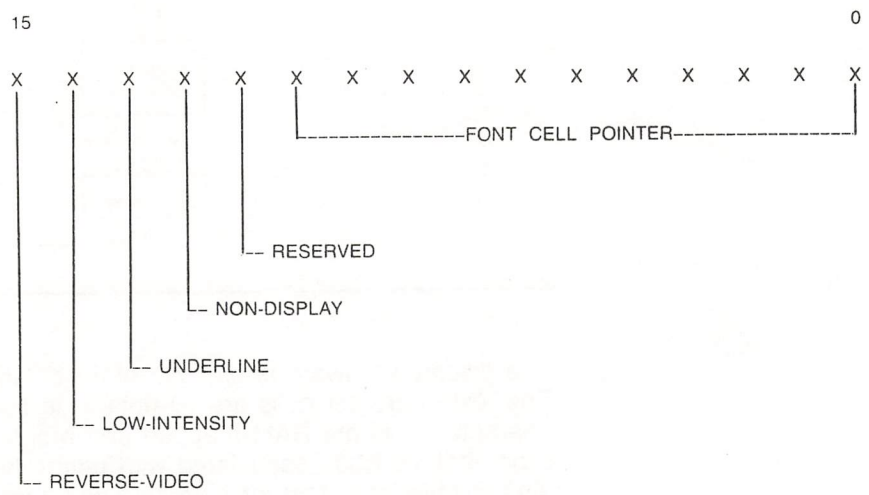


The display hardware is capable of 80 columns by 25 lines of text. The text character cells are 10 dots wide by 16 lines high. These character cells are RAM-mapped and programmable. There is also a 5-bit attribute code associated with each character. Four of these attribute bits are used for reverse-video, underline/strikeover, highlight, and nondisplay. The other bit is available for user software or external hardware. The display hardware can also be configured for a high-resolution mode: 800 by 400 dots of bit-addressable display. In this mode, the reverse-video, double intensity, and nondisplay attributes apply to fixed (16-by-16-dot) cells on the screen, and the underline/strikeover attribute is not operative.

The character and attribute bits are organized into words called the screen buffer. The lower 11 bits of each word define which of the 2048 possible characters is to be placed at that location of the screen. These 11 bits are collectively called the font cell pointer. The upper five bits of the word are the attributes. The MSB (bit 15) is the reverse-video bit. Bit 14 is the low-intensity bit; bit 13 is the underline bit; and bit 12 is the nondisplay bit. The remaining bit is uncommitted.

The screen buffer words are on even-address boundaries. The physical memory of the screen buffer is located, in system address space, at F0000 to F0FFF. The 80-character by 25-line display occupies 2000 words (4000 bytes) of the available 2048 words in the screen buffer. Logically, the screen buffer is mapped to include locations F0000 to F1FFF. Therefore, addressing location F0000 accesses the same physical word as addressing location F1000. The logical beginning of the display screen is selected by a pair of registers in the CRT controller chip (this is a word address). This register pair may be programmed to move the starting address (line one, column one) of the display to any word of the screen buffer. When the control register pair is used in this manner, the screen buffer functions as a 2048-word circular buffer. Using this technique, line scrolling in the text mode may be accomplished by adding 80 to the contents of the screen start register and blanking the 80 words following the previous end of screen. In both these operations, to keep the address within the screen buffer address space, it is also necessary to logically AND the resulting address with F1 FFF.

Figure D-2: Screen Buffer World Format



The actual dot patterns of each character are stored in the font cell memory. Each 10-dot-by-16-line character cell is stored in 16 consecutive words. This group of 16 words is called a font cell. The lower 10 bits of each word contain the 10 dots of a scan line of the character picture. The upper-left bit of a character would be the LSB of the first word in the 16 consecutive words that define a font cell. Bit 15 of each font cell word is reserved for the underline/strikeover flag bit (in text mode, only). If bit 15 is set and the underline/strikeover attribute (bit 13) from the screen buffer is set, then that scan line will be white; otherwise, the lower 10 bits in that word will be displayed. The nondisplay bit can be used to create "secret" (nondisplayed) characters or fields. If a minimum (128-character) set is defined, the font cells would occupy 4K bytes of memory. The font cells can be located anywhere within the first 128K bytes of RAM, but may not cross the 64K boundary.

HIGH RESOLUTION MODE

The 800-by-400-dot, bit-mapped, high-resolution display is a special-case use of the cell graphics. The output line, called HIRES (from the CRT controller), controls the character cell width. When this line is high, the character cells are 16 dots wide instead of the usual 10 dots. The screen is then organized as 50 columns by 25 lines of 16-by-16-dot font cells. This is accomplished by writing new values into the control registers of the CRT controller. The full 16 bits of each font cell word are used to describe the picture of each character. The screen buffer is organized so that each of the 1250 characters on the screen is a different character, as described earlier in this manual. High-resolution software then operates directly on the font cell memory for display bit manipulation.

> Programming Note: The HIRES/TEXT control and the DOTSEL control (which select whether the beginning address of the font cell memory is to be in the first or the second 64K of system memory) are manipulated via the two high-order address bits in the CRTC display address register pair, R12 and R13. This address interacts with the cursor register pair, R14 and R15, and the light pen register pair, R16 and R17. Specifically, if the light pen register pair is used and/or the cursor-display function is desired, then the software must (1) add the cursor address to the current settings of HIRES/TEXT and DOTSEL and (2) subtract or mask these bits when interpreting a light pen interrupt.

BRIGHTNESS AND CONTRAST CONTROL

The overall display brightness and the contrast between high and low intensity characters are software adjustable.

Brightness may be adjusted to one of eight different levels by setting the brightness control bits (PB2, PB3, and PB4 of the 6522 at E8040) to the binary value corresponding to the desired level. The binary-value range from zero to seven selects increasing brightness levels.

The contrast function controls the difference in intensity between highlighted characters and normal intensity characters. Only the intensity of the normal intensity characters is varied by the contrast function. The contrast function selects one of eight levels by setting the binary value of the desired level in the three contrast control bits

(PB5, PB6, and PB7 of the 6522 at E8040). A value range of zero to seven selects increasing differences between the normal and highlighted characters, with zero causing no difference.

CIRCUIT DESCRIPTION

The lower 128K bytes of RAM is a dual-port memory system. One port is used by the display hardware to refresh the raster-scan display. The other port is used by the 8088 microprocessor for read and write operations. The dual-port memory is managed by an arbitrator circuit that guarantees one refresh access to the display RAM every character cell time. The arbitrator circuit adds a wait state to any 8088 memory cycle if this is necessary to isolate it from the display-refresh cycle. This results in an average of one wait state (200 nsec) for every five processor memory access cycles. Processor and memory cycles are normally four clock periods (200 nsec). This could cause a decrease of approximately 5% in system bus performance. However, due to the 8088 instruction lookahead queue, this decrease in bus performance rarely translates into decreased system performance.

The display-refresh addresses are generated by the HD46505S CRT-controller chip (CRTC). Of the 14 address lines from the CRTC, 11 (MA0-MA10) are used to address the 2K words of screen buffer RAM. The 16 data lines output by the screen buffer are latched and divided into 11 lines of character address information and 5 lines of character attributes. The attribute bits are sent, via a set of character sync registers, to the video control section. The 11 lines of the character address are combined with 4 lines of character-row address and MA12 (DOTSEL) from the CRTC. This address is then multiplexed down to 8 font cell address lines. The 14th character address line (MA13) is used to select the high-resolution mode. The 16-bit data output word from each font cell word is latched and sent to a 16-bit shift register. Either 10 or 16 dots of the shift register are shifted out to the video control section. The video control section adds the reverse video, highlight, underline, and nondisplay attribute bits and the cursor output from the CRTC. The result is sent to the video display, along with horizontal and vertical sync pulses.

The display circuit manages the memory refresh in the 128K bytes of on-board dynamic RAM. The horizontal and vertical retrace intervals are used for memory refresh. Display-refresh cycles occuring during retrace intervals cause 8 bits from the refresh-address counter to be sent to all 128K of dynamic RAM, rather than the normal display-address lines. The display CAS signal is inhibited for a RAS-only memory refresh. The memory-refresh counter is clocked after each refresh cycle. In every 64 microsecond horizontal display period, 15 memory-refresh cycles occur. Every 2 ms, 480 memory-refresh addresses are generated, exceeding the 128-address-per-2ms specified requirement of 16K dynamic RAM.

CRTC DEVICE OPERATION

The CRTC consists of an internal register group, horizontal and vertical timing circuits, a linear address generator, a cursor-control circuit, and a light-pen-detection circuit. Horizontal and vertical timing circuits generate RA0-RA4 , DISPTMG, SYNC, and VSYNC. RA0-RA4 are raster (row) address signals and are used as address bits 1 to 4 for the font cell accesses. DISPTMG, HSYNC, and VSYNC signals are sent to the video control circuit. This horizontal and vertical timing circuit consists of an internal counter and comparator circuit.

The linear address generator generates refresh memory address MA0-MA11 to be used for refreshing the screen. The light-pen-detection circuit detects the light pen position on the screen. When the light pen strobe signal is received, the light pen register latches the address generated by the linear address generator to save the position of the pen on the screen. The cursor control circuit controls the position of the cursor, its height, and its blink rate.

The CRTC provides 13 interface signals to the CPU and 25 interface signals to the display circuits.

Table D-1: Recommended Values For CRTC Register Initialization

REGISTER	CHARACTER MODE	HIGH RESOLUTION MODE
R0	5C	3A
R1	50	32
R2	51	34
R3	CF	C9
R4	19	19
R5	06	06
R6	19	19
R7	19	19
R8	03	03
R9	0E	0E
R10	60	20
R11	0F	0F
R12	00	20
R13	00	00
R14	00	00
R15	00	00

NOTE: All values are in hexadecimal.

INTERFACE SIGNALS TO THE CPU

Bidirectional Data Bus (ID0-ID7)

The bidirectional data bus is used for data transfer betweeen the CRTC and the 8088. The data bus outputs are 3-state buffers and remain in the high-impedance state except when the 8088 performs a CRTC read operation.

Read/Write (R/W)

The R/W signal controls the direction of data transfer between the CRTC and the 8088. When R/W is high, CRTC data is transferred to the 8088. When R/W is low, 8088 data is transferred to the CRTC.

Chip Select (CS)

The CS signal is used to address the CRTC. When CS is low, it enables R/W operation to CRTC internal registers. This signal is derived from decoded address signals of the the 8088.

Register Select (RS)

The RS signal is used to select the address register and the 18 control registers of the CRTC. When RS is low, the address register is selected; when RS is high, control registers are selected. This signal is the lowest bit (A0) of the 8088 address bus.

Enable (E)

The E signal is used as strobe signal in 8088 R/W operations with the CRTC internal registers. This signal is PHASE2.

Reset (RES)

The Reset signal (RES) is an input signal used to reset the CRTC. When RES is low, it forces the CRTC into the following status:

- All the counters in the CRTC are cleared, and the device stops the display operation
- All the outputs go low
- Control registers in the CRTC are not affected

INTERFACE SIGNALS TO DISPLAY CIRCUITS

Character Clock (CLK)

CLK is a standard clock input signal which defines character timing for the CRTC display operation. This signal is provided by the memory controller.

Horizontal Sync (HSYNC)

HSYNC is an active high-level signal which provides horizontal synchronization for the display device.

Vertical Sync (VSYNC)

VSYNC is an active high-level signal which provides vertical synchronization for the display device.

Display Timing (DISPTMG)

DISPTMG is an active high-level signal which defines the display period in horizontal and vertical raster scanning. It is necessary to enable the video signal only when DISPTMG is high.

Refresh Memory Address MA0-MA13

MA0-MA11 are refresh memory address signals which are used to access the screen buffer in order to refresh the CRT screen periodically.

MA11 is unused.

MA12 selects the 64K memory bank to be used for font cell memory.

When MA12 equals 0, it selects system RAM starting at location 0; when MA12 equals 1, it selects system RAM starting at location 10000H.

When MA13 equals 0, it selects text mode when MA13 equals one, it selects bit-mapped HIRES mode.